




(<http://blog.oldboyedu.com>)

### 公告

如果你觉得下面的文章晦涩难懂，请猛戳==>[老男孩教育博客博文配套视频 \(http://blog.oldboyedu.com/supporting-video/\)](http://blog.oldboyedu.com/supporting-video/) 

## 你所不知道的TIME\_WAIT和CLOSE\_WAIT

📅 2年前 (2016-03-17)

/ 👤 老男孩 (<http://blog.oldboyedu.com/author/oldboyoldboy/>)

/ 📁 Linux攻略 (<http://blog.oldboyedu.com/category/linux-strategy/>)

/ 💬 2评论 (<http://blog.oldboyedu.com/tcp-wait/#comments>)

/ © 来源：本站原创 / 👁 2849°C / 字体：

小

中

大

你遇到过**TIME\_WAIT**的问题吗？

我相信很多都遇到过这个问题。一旦有用户在喊：网络变慢了。  
第一件事情就是，`netstat -a | grep TIME_WAIT | wc -l`一下。哎  
呀妈呀，几千个**TIME\_WAIT**。

然后，做的第一件事情就是：打开Google或者Bing，输入关键词：`too many time wait`。一定能找到解决方案，而排在最前面或者被很多人到处转载的解决方案一定是：

打开 `sysctl.conf` 文件，修改以下几个参数：

- `net.ipv4.tcp_tw_recycle = 1`
- `net.ipv4.tcp_tw_reuse = 1`
- `net.ipv4.tcp_timestamps = 1`

你也会被告知，开启**tw\_recycle**和**tw\_reuse**一定需要**timestamps**的支持，而且这些配置一般不建议开启，但是对解决**TIME\_WAIT**很多的问题，有很好的用处。

接下来，你就直接修改了这几个参数，**reload**一下，发现，咦，没几分钟，**TIME\_WAIT**的数量真的降低了，也没发现哪个用户说有问题，然后就没有然后了。

做到这一步，相信**50%**或者更高比例的开发就已经止步了。问题好像解决了，但是，要彻底理解并解决这个问题，可能就这么简单，或者说，还有很长的路要走！

### 什么是**TIME-WAIT**和**CLOSE-WAIT**？

所谓，要解决问题，就要先理解问题。随便改两行代码，发现**bug**“没有了”，也不是**bug**真的没有了，只是隐藏在更深的地方，你没有发现，或者以你的知识水平，你无法发现而已。

大家知道，由于**socket**是全双工的工作模式，一个**socket**的关闭，是需要四次握手来完成的。

- 主动关闭连接的一方，调用**close()**；协议层发送**FIN**包
- 被动关闭的一方收到**FIN**包后，协议层回复**ACK**；然后被动关闭的一方，进入**CLOSE\_WAIT**状态，主动关闭的一方等待对方关闭，则进入**FIN\_WAIT\_2**状态；此时，主动关闭的一方等待  
被动关闭一方的应用程序，调用**close**操作
- 被动关闭的一方在完成所有数据发送后，调用**close()**操作；此时，协议层发送**FIN**包给主动关闭的一方，等待对方的**ACK**，被动关闭的一方进入**LAST\_ACK**状态；
- 主动关闭的一方收到**FIN**包，协议层回复**ACK**；此时，主动关闭连接的一方，进入**TIME\_WAIT**状态；而被动关闭的一方，进入**CLOSED**状态
- 等待**2MSL**时间，主动关闭的一方，结束**TIME\_WAIT**，进入**CLOSED**状态

通过上面的一次**socket**关闭操作，你可以得出以下几点：

1. 主动关闭连接的一方 – 也就是主动调用**socket**的**close**操作的一方，最终会进入**TIME\_WAIT**状态

2. 被动关闭连接的一方，有一个中间状态，即CLOSE\_WAIT，因为协议层在等待上层的应用程序，主动调用close操作后才主动关闭这条连接
3. TIME\_WAIT会默认等待2MSL时间后，才最终进入CLOSED状态；
4. 在一个连接没有进入CLOSED状态之前，这个连接是不能被重用的！

所以，这里凭你的直觉，TIME\_WAIT并不可怕（not really，后面讲），CLOSE\_WAIT才可怕，因为CLOSE\_WAIT很多，表示说要么是你的应用程序写的有问题，没有合适的关闭socket；要么是说，你的服务器CPU处理不过来（CPU太忙）或者你的应用程序一直睡眠到其它地方(锁，或者文件I/O等等)，你的应用程序获得不到合适的调度时间，造成你的程序没法真正的执行close操作。

这里又出现两个问题：

5. 上文提到的连接重用，那连接到底是个什么概念？
6. 协议层为什么要设计一个TIME\_WAIT状态？这个状态为什么默认等待2MSL时间才会进入CLOSED

先解释清楚这两个问题，我们再来看，开头提到的几个网络配置究竟有什么用，以及TIME\_WAIT的后遗症问题。

**Socket**连接到底是个什么概念？

大家经常提socket，那么，到底什么是一个socket？其实，socket就是一个五元组，包括：

7. 源IP
8. 源端口
9. 目的IP
10. 目的端口
11. 类型：TCP or UDP

这个五元组，即标识了一条可用的连接。注意，有很多人把一个socket定义成四元组，也就是源IP:源端口 + 目的IP:目的端口，这个定义是不正确的。

例如，如果你的本地出口IP是180.172.35.150，那么你的浏览器在连接某一个Web服务器，例如百度的时候，这条socket连接的四元组可能就是：

[180.172.35.150:45678, tcp, 180.97.33.108:80]

源IP为你的出口IP地址 180.172.35.150，源端口为随机端口 45678，目的IP为百度的某一个负载均衡服务器IP 180.97.33.108，端口为HTTP标准的80端口。

如果这个时候，你再开一个浏览器，访问百度，将会产生一条新的连接：

[180.172.35.150:43678, tcp, 180.97.33.108:80]

这条新的连接的源端口为一个新的随机端口 43678。

如此来看，如果你的本机需要压测百度，那么，你最多可以创建多少个连接呢？我在文章《云思路 | 轻松构建千万级投票系统》里也稍微提过这个问题，没有阅读过本文的，可以发送“投票系统”阅读。

第二个问题，**TIME\_WAIT**有什么用？

如果我们来做个类比的话，**TIME\_WAIT**的出现，对应的是你的程序里的异常处理，它的出现，就是为了解决网络的丢包和网络不稳定所带来的其他问题：

第一，防止前一个连接【五元组，我们继续

以 180.172.35.150:45678, tcp, 180.97.33.108:80 为例】

上延迟的数据包或者丢失重传的数据包，被后面复用的连接

【前一个连接关闭后，此时你再次访问百度，新的连接可能还是由180.172.35.150:45678, tcp, 180.97.33.108:80 这个五元组来表示，也就是源端口凑巧还是45678】错误的接收（异常：数据丢了，或者传输太慢了），参见下图：

•

- SEQ=3的数据包丢失，重传第一次，没有得到ACK确认
- 如果没有TIME\_WAIT，或者TIME\_WAIT时间非常短，那么关闭的连接【180.172.35.150:45678, tcp, 180.97.33.108:80 的状态变为了CLOSED，源端口可被再次利用】，马上被重用【对180.97.33.108:80新建的连接，复用了之前的随机端口45678】，并连续发送SEQ=1,2 的数据包
- 此时，前面的连接上的SEQ=3的数据包再次重传，同时，seq的序号刚好也是3（这个很重要，不然，SEQ的序号对不上，就会RST掉），此时，前面一个连接上的数据被后面的一个连接错误的接收

第二，确保连接方能在时间范围内，关闭自己的连接。其实，也是因为丢包造成的，参见下图：

•

- 主动关闭方关闭了连接，发送了FIN；
- 被动关闭方回复ACK同时也执行关闭动作，发送FIN包；此时，被动关闭的一方进入LAST\_ACK状态
- 主动关闭的一方回去了ACK，主动关闭一方进入TIME\_WAIT状态；
- 但是最后的ACK丢失，被动关闭的一方还继续停留在LAST\_ACK状态
- 此时，如果没有TIME\_WAIT的存在，或者说，停留在TIME\_WAIT上的时间很短，则主动关闭的一方很快就进入了CLOSED状态，也即是说，如果此时新建一个连接，源随机端口如果被复用，在connect发送SYN包后，由于被动方仍认为这条连接【五元组】还在等待ACK，但是却收到了SYN，则被动方会回复RST
- 造成主动创建连接的一方，由于收到了RST，则连接无法成功

所以，你看到了，**TIME\_WAIT**的存在是很重要的，如果强制忽略**TIME\_WAIT**，还是有很高的机率，造成数据粗乱，或者短暂性的连接失败。

那么，为什么说，**TIME\_WAIT**状态会是持续**2MSL**（2倍的**max segment lifetime**）呢？这个时间可以通过修改内核参数调整吗？第一，这个**2MSL**，是**RFC 793**里定义的，参见**RFC**的截图标红的部分：

这个定义，更多的是一种保障（**IP**数据包里的**TTL**，即数据最多存活的跳数，真正反应的才是数据在网络上的存活时间），确保最后丢失了**ACK**，被动关闭的一方再次重发**FIN**并等待回复的**ACK**，一来一去两个来回。内核里，写死了这个**MSL**的时间为：**30秒**（有读者提醒，**RFC**里建议的**MSL**其实是**2分钟**，但是很多实现都是**30秒**），所以**TIME\_WAIT**的即为**1分钟**：

所以，再次回想一下前面的问题，如果一条连接，即使在四次握手关闭了，由于**TIME\_WAIT**的存在，这个连接，在**1分钟**之内，也无法再次被复用，那么，如果你用一台机器做压测的客户端，你一分钟能发送多少并发连接请求？如果这台是一个负载均衡服务器，一台负载均衡服务器，一分钟可以有多少个连接同时访问后端的服务器呢？

**TIME\_WAIT**很多，可怕吗？

如果你通过 `ss -tan state time-wait | wc -l` 发现，系统中有很多**TIME\_WAIT**，很多人都会紧张。多少算多呢？几百几千？如果是这个量级，其实真的没必要紧张。第一，这个量级，因为**TIME\_WAIT**所占用的内存很少很少；因为记录和寻找可用的**local port**所消耗的CPU也基本可以忽略。

会占用内存吗？当然！任何你可以看到的数据，内核里都需要有相关的数据结构来保存这个数据啊。一条Socket处于TIME\_WAIT状态，它也是一条“存在”的socket，内核里也需要有保持它的数据：

1. 内核里有保存所有连接的一个hash table，这个hash table里面既包含TIME\_WAIT状态的连接，也包含其他状态的连接。主要用于有新的数据到来的时候，从这个hash table里快速找到这条连接。不同的内核对这个hash table的大小设置不同，你可以通过dmesg命令去找到你的内核设置的大小：  
•
2. 还有一个hash table用来保存所有的bound ports，主要用于可以快速的找到一个可用的端口或者随机端口：  
•

由于内核需要保存这些数据，必然，会占用一定的内存。

会消耗CPU吗？当然！每次找到一个随机端口，还是需要遍历一遍bound ports的吧，这必然需要一些CPU时间。

TIME\_WAIT很多，既占内存又消耗CPU，这也是为什么很多人，看到TIME\_WAIT很多，就蠢蠢欲动的想去干掉他们。其实，如果你再进一步去研究，1万条TIME\_WAIT的连接，也就多消耗1M左右的内存，对现代的很多服务器，已经不算什么了。至于CPU，能减少它当然更好，但是不至于因为1万多个hash item就担忧。

如果，你真的想去调优，还是需要搞清楚别人的调优建议，以及调优参数背后的意义！

## TIME\_WAIT调优，你必须理解的几个调优参数

在具体的图例之前，我们还是先解析一下相关的几个参数存在的意义。

## net.ipv4.tcp\_timestamps

RFC 1323 在 TCP Reliability一节里，引入了timestamp的TCP option，两个4字节的时间戳字段，其中第一个4字节字段用来保存发送该数据包的时间，第二个4字节字段用来保存最近一次接收对方发送到数据的时间。有了这两个时间字段，也就有了后续优化的余地。

tcp\_tw\_reuse 和 tcp\_tw\_recycle就依赖这些时间字段。

## net.ipv4.tcp\_tw\_reuse

字面意思，reuse TIME\_WAIT状态的连接。

时刻记住一条socket连接，就是那个五元组，出现TIME\_WAIT状态的连接，一定出现在主动关闭连接的一方。所以，当主动关闭连接的一方，再次向对方发起连接请求的时候（例如，客户端关闭连接，客户端再次连接服务端，此时可以复用了；负载均衡服务器，主动关闭后端的连接，当有新的HTTP请求，负载均衡服务器再次连接后端服务器，此时也可以复用），可以复用TIME\_WAIT状态的连接。

通过字面解释，以及例子说明，你看到了，tcp\_tw\_reuse应用的场景：某一方，需要不断的通过“短连接”连接其他服务器，总是自己先关闭连接(TIME\_WAIT在自己这方)，关闭后又不断的重新连接对方。

那么，当连接被复用了之后，延迟或者重发的数据包到达，新的连接怎么判断，到达的数据是属于复用后的连接，还是复用前的连接呢？那就需要依赖前面提到的两个时间字段了。复用连接后，这条连接的时间被更新为当前的时间，当延迟的数据达到，延迟数据的时间是小于新连接的时间，所以，内核可以通过时间判断出，延迟的数据可以安全的丢弃掉了。



这个配置，依赖于连接双方，同时对timestamps的支持。同时，这个配置，仅仅影响outbound连接，即做为客户端的角色，连接服务端[connect(dest\_ip, dest\_port)]时复用TIME\_WAIT的socket。

### net.ipv4.tcp\_tw\_recycle

字面意思，销毁掉 TIME\_WAIT。

当开启了这个配置后，内核会快速的回收处于TIME\_WAIT状态的socket连接。多快？不再是2MSL，而是一个RTO（retransmission timeout，数据包重传的timeout时间）的时间，这个时间根据RTT动态计算出来，但是远小于2MSL。

有了这个配置，还是需要保障丢失重传或者延迟的数据包，不会被新的连接(注意，这里不再是复用了，而是之前处于TIME\_WAIT状态的连接已经被destroy掉了，新的连接，刚好是和某一个被destroy掉的连接使用了相同的五元组而已)所错误的接收。在启用该配置，当一个socket连接进入TIME\_WAIT状态后，内核里会记录包括该socket连接对应的五元组中的对方IP等在内的一些统计数据，当然也包括从该对方IP所接收到的最近的一次数据包时间。当有新的数据包到达，只要时间晚于内核记录的这个时间，数据包都会被统统的丢掉。

这个配置，依赖于连接双方对timestamps的支持。同时，这个配置，主要影响到了inbound的连接（对outbound的连接也有影响，但是不是复用），即做为服务端角色，客户端连进来，服务端主动关闭了连接，TIME\_WAIT状态的socket处于服务端，服务端快速的回收该状态的连接。

由此，如果客户端处于NAT的网络(多个客户端，同一个IP出口的网络环境)，如果配置了tw\_recycle，就可能在一个RTO的时间内，只能有一个客户端和自己连接成功(不同的客户端发包的时间不一致，造成服务端直接把数据包丢弃掉)。

我尽量尝试用文字解释清楚，但是，来点案例和图示，应该有助于我们彻底理解。

我们来看这样一个网络情况：

•

3. 客户端IP地址为：180.172.35.150，我们可以认为是浏览器
4. 负载均衡有两个IP，外网IP地址为 115.29.253.156，内网地址为10.162.74.10；外网地址监听80端口
5. 负载均衡背后有两台Web服务器，一台IP地址为10.162.74.43，监听80端口；另一台为 10.162.74.44，监听 80 端口
6. Web服务器会连接数据服务器，IP地址为 10.162.74.45，监听 3306 端口

这种简单的架构下，我们来看看，在不同的情况下，我们今天谈论的tw\_reuse/tw\_recycle对网络连接的影响。

先做个假定：

7. 客户端通过HTTP/1.1连接负载均衡，也就是说，HTTP协议投Connection为keep-alive，所以我们假定，客户端对负载均衡服务器的socket连接，客户端会断开连接，所以，TIME\_WAIT出现在客户端
8. Web服务器和MySQL服务器的连接，我们假定，Web服务器上的程序在连接结束的时候，调用close操作关闭socket资源连接，所以，TIME\_WAIT出现在 Web 服务器端。

那么，在这种假定下：

9. Web服务器上，肯定可以配置开启的配置：tcp\_tw\_reuse；如果Web服务器有很多连向DB服务器的连接，可以保证socket连接的复用。

## 10. 那么，负载均衡服务器和Web服务器，谁先关闭连接，则决定了我们怎么配置tcp\_tw\_reuse/tcp\_tw\_recycle了

### 方案一：负载均衡服务器

首先关闭连接

在这种情况下，因为负载均衡服务器对Web服务器的连接，TIME\_WAIT大都出现在负载均衡服务器上，所以，在负载均衡服务器上的配置：

- net.ipv4.tcp\_tw\_reuse = 1 //尽量复用连接
- net.ipv4.tcp\_tw\_recycle = 0 //不能保证客户端不在NAT的网络啊

在Web服务器上的配置为：

- net.ipv4.tcp\_tw\_reuse = 1 //这个配置主要影响的是Web服务器到DB服务器的连接复用
- net.ipv4.tcp\_tw\_recycle:  
设置成1和0都没有任何意义。想一想，在负载均衡和它的连接中，它是服务端，但是TIME\_WAIT出现在负载均衡服务器上；它和DB的连接，它是客户端，recycle对它并没有什么影响，关键是reuse

### 方案二：Web服务器首先关闭来自负载均衡服务器的连接

在这种情况下，Web服务器变成TIME\_WAIT的重灾区。负载均衡对Web服务器的连接，由Web服务器首先关闭连接，TIME\_WAIT出现在Web服务器上；Web服务器对DB服务器的连接，由Web服务器关闭连接，TIME\_WAIT也出现在它身上，此时，负载均衡服务器上的配置：

- net.ipv4.tcp\_tw\_reuse: 0 或者 1 都行，都没有实际意义
- net.ipv4.tcp\_tw\_recycle=0 //一定是关闭recycle

在Web服务器上的配置：

- net.ipv4.tcp\_tw\_reuse = 1 //这个配置主要影响的是Web服务器到DB服务器的连接复用
- net.ipv4.tcp\_tw\_recycle=1 //由于在负载均衡和Web服务器之间并没有NAT的网络，可以考虑开启recycle，加速由于负载均衡和Web服务器之间的连接造成的大量TIME\_WAIT

回答几个大家提到的几个问题

1. 请问我们所说连接池可以复用连接，是不是意味着，需要等到上个连接time wait结束后才能再次使用？

所谓连接池复用，复用的一定是活跃的连接，所谓活跃，第一表明连接池里的连接都是ESTABLISHED的，第二，连接池做为上层应用，会有定时的心跳去保持连接的活跃性。既然连接都是活跃的，那就不存在有TIME\_WAIT的概念了，在上篇里也有提到，TIME\_WAIT是在主动关闭连接的一方，在关闭连接后才进入的状态。既然已经关闭了，那么这条连接肯定已经不在连接池里面了，即被连接池释放了。

2. 想请问下，作为负载均衡的机器随机端口使用完的情况下大量time\_wait，不调整你文字里说的那三个参数，有其他的更好的方案吗？

第一，随机端口使用完，你可以通过调整/etc/sysctl.conf下的net.ipv4.ip\_local\_port\_range配置，至少修改成net.ipv4.ip\_local\_port\_range=1024 65535，保证你的负载均衡服务器至少可以使用6万个随机端口，也即可以有6万的反向代理到后端的连接，可以支持每秒1000的并发（想一想，因为TIME\_WAIT状态会持续1分钟后消失，所以一分钟最多有6万，每秒1000）；如果这么多端口都使用完了，也证明你应该加服务器了，或者，你的负载均衡服务器需要配置多个IP地址，或者，你的后端服务器需要监听更多的端口和配置更多的IP（想一下socket的五元组）

第二，大量的TIME\_WAIT，多大量？如果是几千个，其实不用担心，因为这个内存和CPU的消耗有一些，但是是可以忽略的。

第三，如果真的量很大，上万上万的那种，可以考虑，让后端的服务器主动关闭连接，如果后端服务器没有外网的连接只有负载均衡服务器的连接（主要是没有NAT网络的连接），可以在后端服务器上配置tw\_recycle，然后同时，在负载均衡服务器上，配置tw\_reuse。

### 3. 如果想深入的学习一下网络方面的知识，有什么推荐的？

学习网络比学一门编程语言“难”很多。所谓难，其实，是因为需要花很多的时间投入。我自己不算精通，只能说入门和理解。基本书可以推荐：《TCP/IP 协议详解》，必读；

《TCP/IP 高效编程：改善网络程序的44个技巧》，必读；

《Unix环境高级编程》，必读；《Unix网络编程：卷一》，我只读过卷一；另外，还需要熟悉一下网络工具，tcpdump以及wireshark，我的notes里有一个一站式学

习Wireshark：<https://github.com/dafang/notebook/issues/114>，也值得一读。有了这些积累，可能就是一些实践以及碎片化的学习和积累了。

写在最后

这篇文章我断断续续写了两三天，内容找了多个地方去验证，包括看到Vincent Bernat的一篇文章以及Vincent在多个地方和别人的讨论。期间，我也花了一些时间和Vincent探讨了几个我没在tcp源码里翻找到的有疑问的地方。

我力求比散布在网上的文章做到准确并尽量整理的清晰一些。

但是，也难免会

有疏漏或者有错误的地方，高手看到可以随时指正，并和我讨论，大家一起研究！

感谢您阅读。

原文链接 ([http://mp.weixin.qq.com/s?](http://mp.weixin.qq.com/s?__biz=MzI4MjA4ODU0Ng==&mid=402415747&idx=1&sn=2458ba4fe1830eecd8b1)

[\\_\\_biz=MzI4MjA4ODU0Ng==&mid=402415747&idx=1&sn=2458ba4fe1830eecd8b1](http://mp.weixin.qq.com/s?__biz=MzI4MjA4ODU0Ng==&mid=402415747&idx=1&sn=2458ba4fe1830eecd8b1)

AD：官方群:运维交流09群385168604 Linux交流QQ群339128815  
(<http://oldboy.blog.51cto.com/>)

👍 赞16

如无特殊说明，文章均为本站原创，转载请注明出处

- 转载请注明来源：你所不知道的TIME\_WAIT和CLOSE\_WAIT (<http://blog.oldboyedu.com/tcp-wait/>)
- 本文永久链接地址：<http://blog.oldboyedu.com/tcp-wait/>

该文章由 老男孩

(<http://blog.oldboyedu.com/author/oldboyoldboy/>)

发布



这货来去如风，什么鬼都没留下！！



« 服务端高性能数据库优化演变细节案例 (<http://blog.oldboyedu.co>

清华申请退学博士作品：完全用Linux工作 (<http://blog.oldboyedu.c>



发表我的评论

Hi，请填写昵称和邮箱！

客官、确定不来一发嘛...



</> 代码 贴图 B 加粗 链接 删除线 签到

☐ 有人回复时邮件通知我

发射...

🗨 (2)条精彩评论:



第一遍看的时候，比较懵，时隔一个月，懂了点什么。 😁

daxin 2016-05-28 09:20 回复

(<http://blog.oldboyedu.com/tcp-wait/?replytocom=54#respond>)



看第一遍，有点懵懂；看第二遍，把4次握手图画好，对照着看，脑子思路就很清晰了，感谢老师无私奉献

xiaoq114 2017-01-10 10:27 回复

(<http://blog.oldboyedu.com/tcp-wait/?replytocom=114#respond>)

Copyright © 2018 老男孩教育博客 (<http://blog.oldboyedu.com>)  
All rights reserved. | 基于 WordPress (<https://cn.wordpress.org/>)  
& 阿里云 (<http://www.aliyun.com/>)